Report for Wack

Viktor Franzen, Tobias Lindroth, Spondon Siddiqui, Alexander Solberg

October 25, 2018

Contents

1 Introduction

The purpose of the project is to design and create a messaging application in which multiple users can communicate with another. To differentiate from other messaging applications, the main purpose is that multiple users can join a group to communicate primarily on desktops or laptops. Students working in groups will benefit from the application as it provides a simple, private space for them to communicate and share information. As group work is common within education, the application serves a purpose for higher education and is therefore required to simplify communication between groups.

The application will be a cross-platform desktop application with a graphical user interface. The aim is develop the application so it is easy and pleasant to use, as well as have the possibility of being extended into a more complex messaging application. First time users get the possibility to enter the application, identify themselves, and thereafter search for groups they wish to join and communicate with. Once active, the user can view, send, and receive messages.

1.1 Definitions, acronyms, and abbreviations

GUI: Graphical User Interface; How the application looks.

User Story: An informal description of a feature of a software system, written in natural language. Often written from the perspective of an end user.

MVC: Model-View-Controller; A design pattern that separates the logic from the view which makes it easy to reuse the model or change view.

UML: Unified Modeling Language; A general-purpose, modeling language used in the development of object oriented projects. It provides a standard way for the design of a system to be visualized.

JSON: JavaScript Object Notation; A language independent data format.

JavaFX: A Java standard GUI library.

FXML: An XML-based user interface markup language.

Channel: A chat group; A group where users of the application can join and send messages. Other users in the same group can see those messages.

2 Requirements

2.1 User Stories

Story Identifier: STR00 Story Name: Send message

Description

As a user I want to be able to send a message to a channel so that other users in the channel can see it.

Acceptance criteria

Functional

- A user can send a message to a channel
- A user can't send an empty message to a channel
- There is a field where the user can write messages
- There is a send button
- All group members receive messages from the channel

Non-functional

- Security - Are users that are not members of a channel prevented from seeing the channel's messages?
- Documentation - Is the code well documented?
- Testability - Can the code be tested in some way?

Story Identifier: STR03 Story Name: Create public group

Description

As a user I want to be able to create a public group conversation so that multiple people can talk together

Acceptance criteria

Functional

- There is a button the user can press to create a new group
- There is a field where the user can give the group a name
- There is a field where the user can give the group a description
- Other users can join the group

Non-functional

- Response time - Can other users join the channel directly after it is created?
- Documentation - Is the code well documented?
- Testability - Can the code be tested in some way?

Story Identifier: STR02 Story Name: Use Identification

Description

As a user I want to be able to use identification so the other users know who I am.

Acceptance criteria

Functional

- I can choose an identification
- Other people can see my identification

Non-functional

- Security - Two users can't have the same identification
- Documentation - Is the code well documented?
- Testability - Can the code be tested in some way?

Story Identifier: STR01 Story Name: Git repository

Description

As a team we want everyone to have the same Git repository so we can work on the same project without conflicts

Acceptance criteria

Functional

- There is a Git repository online
- Each group member has access to the repository

- Each group member understands how to use Git

Story Identifier: STR09 Story Name: Overview of channels

Description

As a user I constantly want to to see an overview of all my conversations so i can keep track of the channels I'm active in.

Acceptance criteria

Functional

- There is a list with the channels the user is a member in
- The list is updated when joining a channel
- The list is update when leaving a channel

Non-functional

- Documentation - Is the code well documented?
- Testability - Can the code be tested in some way?

Story Identifier: STR11 Story Name: Find channels

Description

As a user I want to be able to see public channels with a specific topic so i can join a channel I'm interested in.

Acceptance criteria

Functional

- There is a field where you can search for channels
- There is a join button for channels in the search result
- You become a member of the channel when the join button is pressed

Non-functional

- Response time - The search process is fast and the results are displayed immediately.
- Documentation - Is the code well documented?
- Testability - Can the code be tested in some way?

Story Identifier: STR18 Story Name: Save Channel Data

Description

As a user I want my data and messages to be saved somewhere so it exists if I need it again

Acceptance criteria

Functional

- The user's account is saved
- The user can log into an existing account
- The users data remains unchanged after login
- The user only sees their own data
- The user doesn't actively have to save the data

Non-functional

- Security - Can the password be matched without revealing the data?
- Capacity - How much data can the system store and for how long?
- Reliability - Is the data transferred in a reliable way?
- Documentation - Is the code well documented?
- Testability - Can the code be tested in some way?

Story Identifier: STR08 Story Name: See old messages

Description

As a user I want to be able to look back at a conversation so I can view information I've forgotten

Acceptance criteria

Functional

- The messages are saved
- The user can view the messages
- The messages return to the correct conversation
- The messages remain in the correct order
- The user doesn't have to actively save the messages

Non-functional

- Reliability - Is the data transferred in a reliable way?
- Security - Are users that are not members of a channel prevented from seeing the channel's messages?
- Documentation - Is the code well documented?
- Testability - Can the code be tested in some way?

Story Identifier: STR04 Story Name: Leave channel

Description

As a channel member I want to be able to leave a channel conversation if needed so that I don't receive unwanted messages

Acceptance criteria

Functional

- There is a button the user can press to leave a channel
- Other users in the channel get an indication that a user has left

Non-functional

- Security - The user who left can no longer see that channel's messages
- Documentation - Is the code well documented?
- Testability - Can the code be tested in some way?

Story Identifier: STR05 Story Name: Kick channelmember

Description

As an administrator I want to be able to kick people from a channel if needed so that unwanted people cannot see my private conversations.

Acceptance criteria

Functional

- There is a list with all members of the group
- The administrator can choose a member and press a button to kick the chosen member
- The member who got kicked gets an indication that he/she has been kicked
- The former member can't see the groups messages anymore

Non-functional

- Response time - The search process is fast and the results are displayed immediately.
- Documentation - Is the code well documented?
- Testability - Can the code be tested in some way?

Story Identifier: STR07 Story Name: New message notification

Description

As a user I want an indication that I've received a new message so that I don't miss any messages.

Acceptance criteria

Functional

- Groups with non-read messages are "higlighted" in some way
- The marking disappears when the user has read the message

Non-functional

- Documentation - Is the code well documented?
- Testability - Can the code be tested in some way?

2.2 User interface

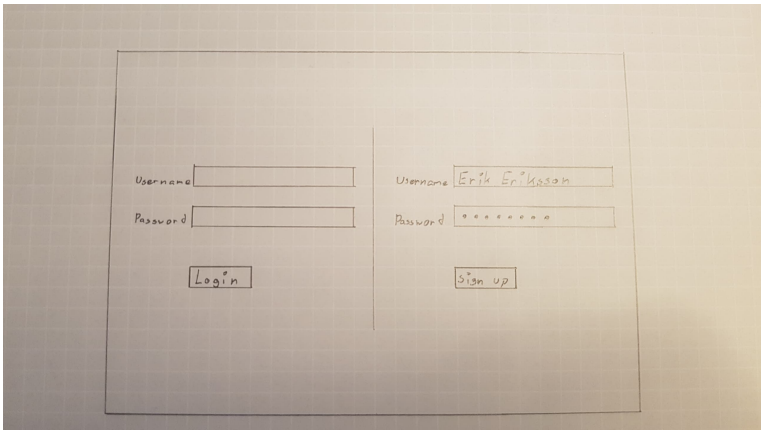


Figure 1: The log-in screen version 1

When the application is started the user is met with a view for logging in or signing up. If the user has an existing account, they can can use the fields to the left to log in to it. If they wish to create a new

account, they can do so using the fields to the right. The user has to type in a username and a password for either of these to work.

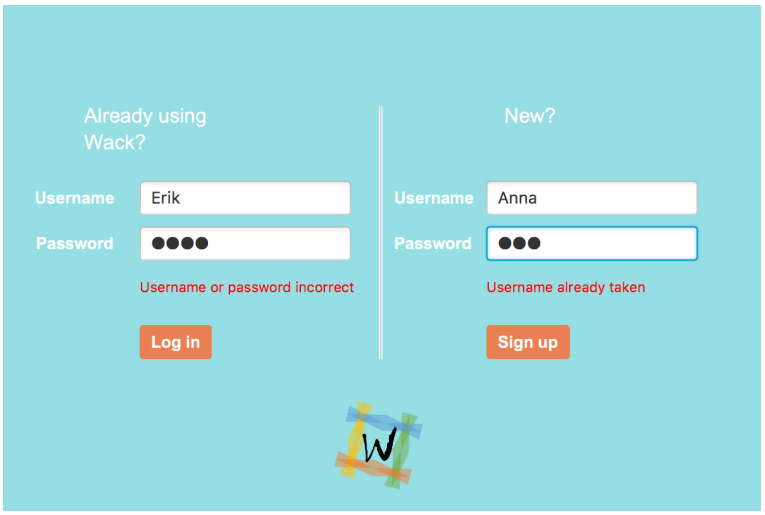


Figure 2: The log-in screen version 2

This is the second iteration of the login view. Color has been added to the background, buttons and text. The view now displays an error message if the user has inputted an incorrect username or password, or if they try to create a user with a name that is already taken.

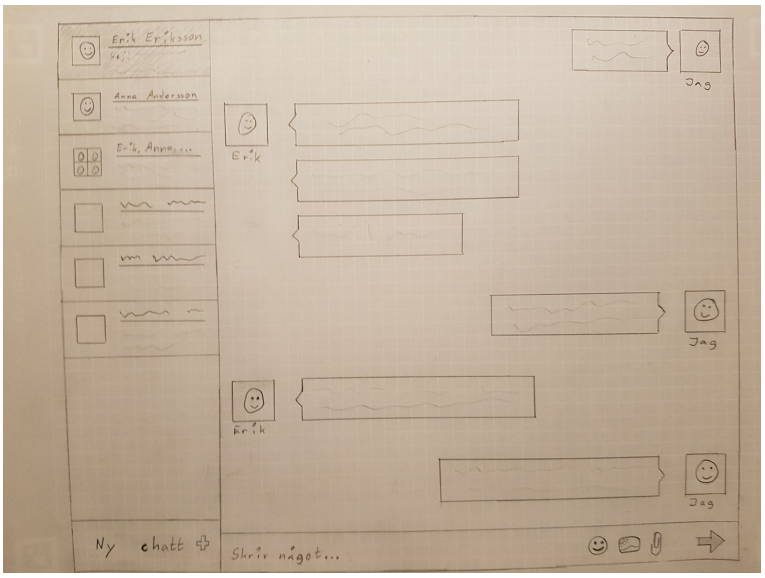


Figure 3: The main view version 1

This is the first iteration of the main view, which will show after the user has logged in or signed up. Here, they can view the channels they are in to the left, and the channel they are currently in is shown to the right/center of the screen. Here, the history of the channel's messages are shown, and the messages are displayed next to the profile picture of the user who sent them. The field at the bottom of the screen is used for typing in messages, and the arrow-button in the bottom-right corner is used for sending messages. The button in the bottom-left corner is used for creating a new channel.

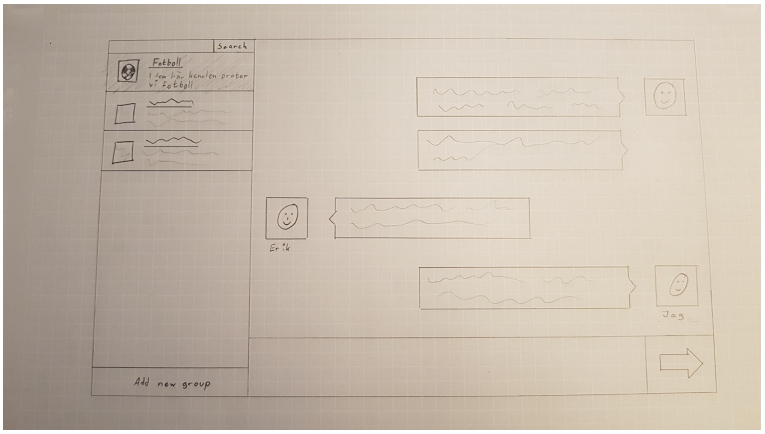


Figure 4: The main view version 2

This is the second iteration of the main view. The application has been made wider and less squarish and the channels are now displayed differently. Now, the name and the description of the channel is shown, as opposed to the users and the latest messages of the channel. Furthermore, a searchbar has been added to the top-left corner to allow users to search for a desired channel.

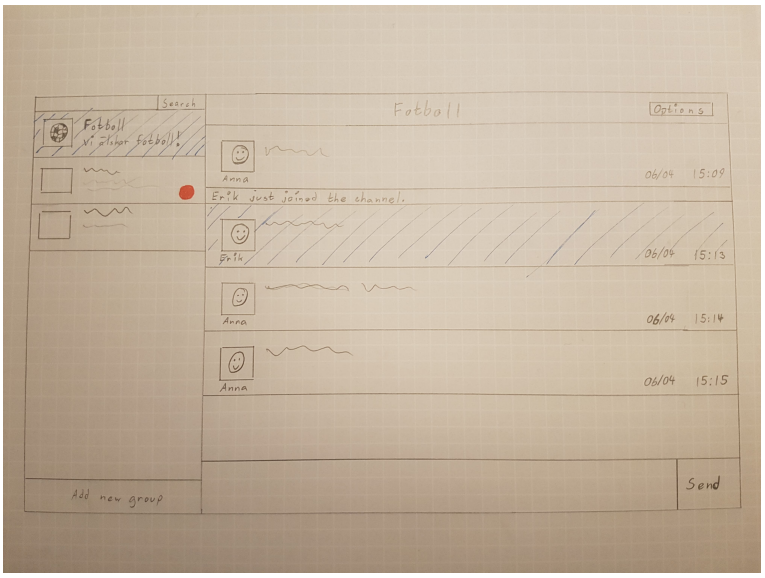


Figure 5: The main view version 3

This is the third iteration of the main view. The way the messages are shown has been simplified to make it more intuitive for the user. The messages now more clearly display who has said what. In addition, a light blue color has been added to signify which channel is selected, and which messages the user has written.

The channel now notifies channel members when a user has joined or left the channel. These messages are disparate from regular user-written messages in appearance; they are thinner and don't have a user linked to them. This makes it easier to differentiate between notifications and messages. The messages now also display the time and date of their sending.

A banner has been added to the top with the name of the current channel. Furthermore, a red dot is now displayed whenever the user has unread messages in another channel, so the user can keep themselves up to date with all of their channels. Lastly, an options button has been added to the top-right corner of the application, which lets the user leave the channel.

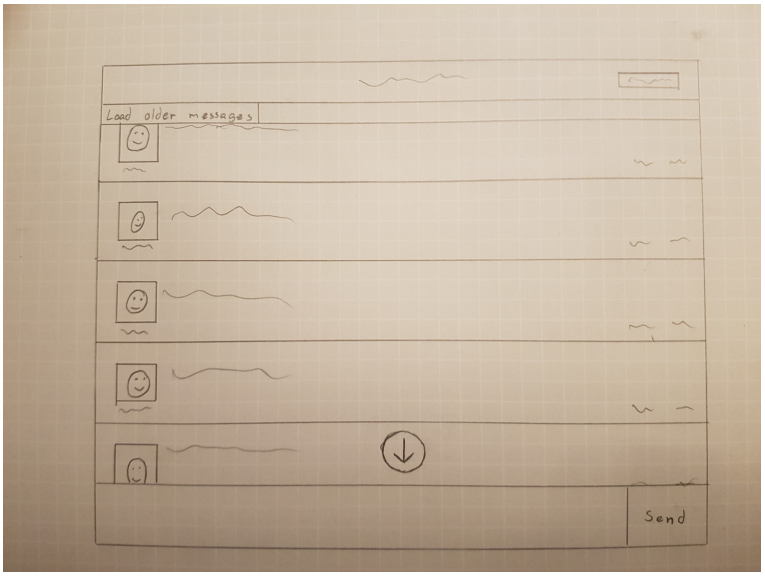


Figure 6: Messages from a single channel

Two new functions has been added to the application. One is a button at the top of the view, just beneath the banner, when the user has scrolled up to the oldest loaded message. This button let's the user load older messages, if they should choose to do so. This is because the application does not load in all the messages in a channel at once, out of convenience for the user; more often than not, the user does not wish to see messages from long ago. However, if they would want to, the possibility is there.

Additionally, there is now a circular button with an arrow at the bottom of the application. This button is shown whenever the user has unseen messages in the current channel, perhaps because they are viewing older messages. This button puts the user at the bottom of the channels messages, as the application does not scroll down automatically.

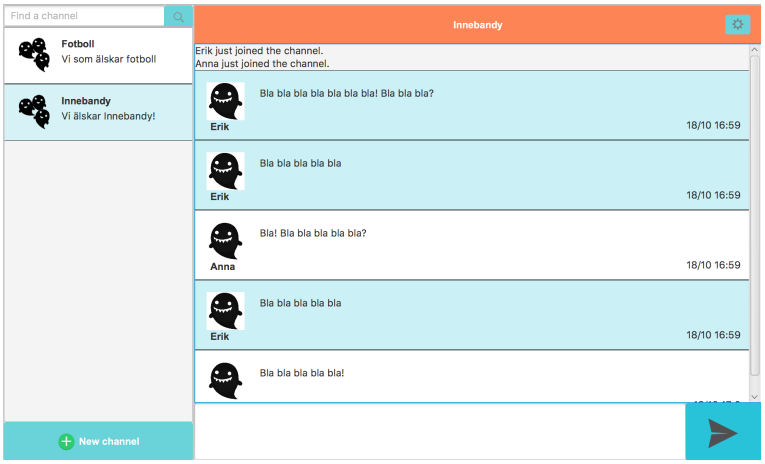


Figure 7: The main view version 4

This is the fourth iteration of the main view. A color scheme has been added to the entire application; the colors light blue and orange, which are complementary and therefore fit well together. The buttons now use symbols instead of text and stick out thanks to their color. The options button now include a list of the channels members, and if a user is the administrator of the group, a button to kick unwanted members.

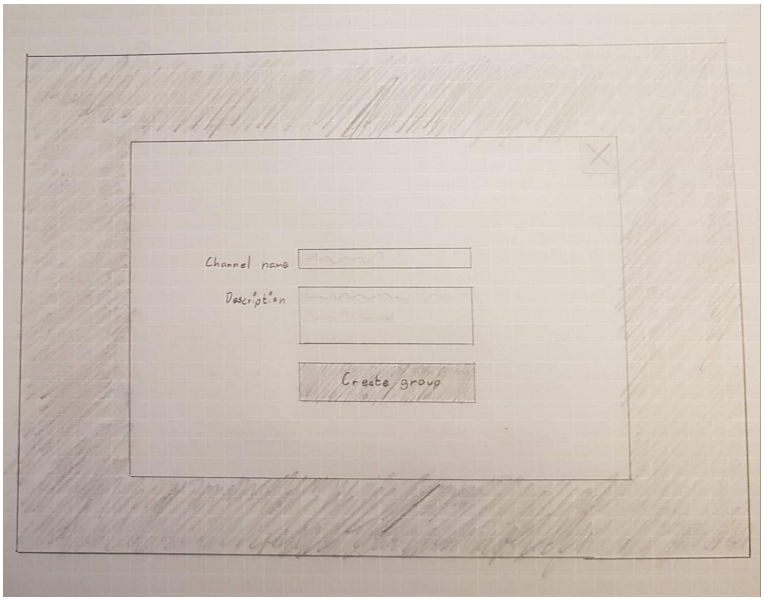


Figure 8: Creating a new channel

This is the view for creating new channels. The view is shown when the user presses the "Add new group" button at the bottom-left of the main view, and is shown on top of the main view. Moreover the main view is darkened to highlight the creation-view. In order to create a new channel, the user has to type in a name and a description for the channel. The user can choose to cancel the creation by either clicking on the x-button in the top-right corner, or by clicking outside on the view, on the main view behind it. The user is then brought back to the main view.

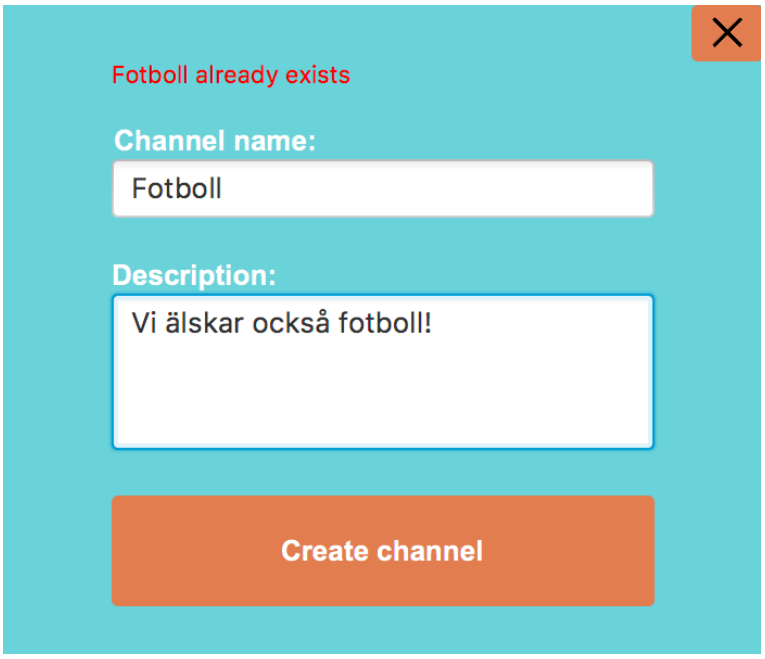


Figure 9: Creating a new channel version 2

The box for creating new channels has been colored according to the color scheme and the entire box has been made smaller. The view now also displays an error message if the user tries to create a channel with a name that already exists.

3 Domain model

The domain model consists of five objects. Server, channel, user, client and message. Together, they form the core of the application.

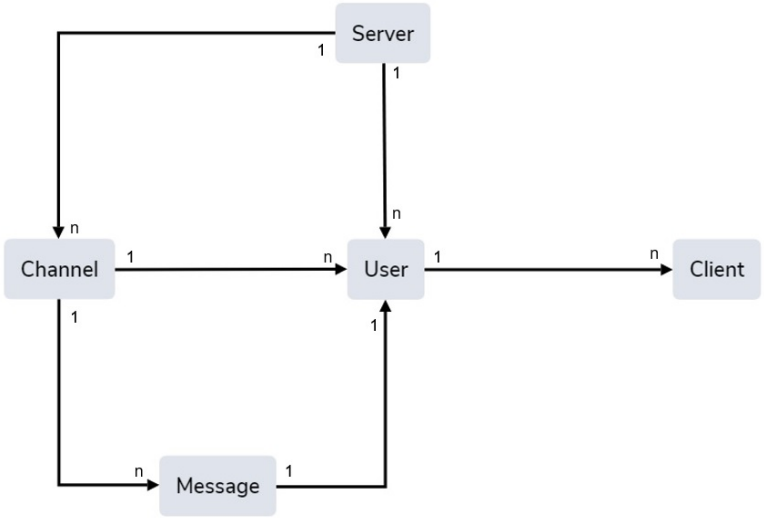


Figure 10: Domain model

3.1 Class responsibilities

The server class is responsible for keeping track of all the existing users and channels.

The channel class holds all the messages written to it and keeps track of all the members. You can use the channel to send a message, and then the channel is responsible for delivering this message to all the members of the channel.

The user class is responsible of keeping track of its clients and forwarding all the received messages to the clients.

The client is responsible for being observable and forwarding any messages it receives to any eventual listener.

The message class is responsible for all the information regarding a message. The message content, who sent it, and when the message was created.

4 System Architecture

This chat application will simulate an actual chat application with multiple computers. Hence, everything will be on the same computer, but using multiple clients.

When starting the program, there will be a login/sign up screen, from which the user can create clients. A client will open in a separate window and then the login/sign up screen can be used to create more clients.

In the top level of this application, we have the model, controllers, views and services packages and the main class (lies in ”default package” in figure). There are no circular dependencies, and as seen in the

figure below, the controller and services package both uses the model, but the model is not aware of who is using it.

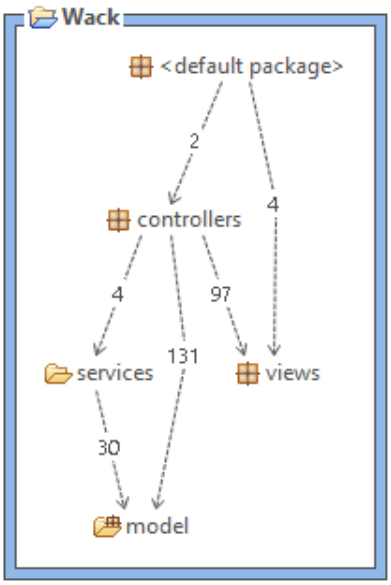


Figure 11: The top level

A MVC pattern has been used to structure the application. Hence, the application is split into model, view, and controller.

The model is not aware of what entity is using its data and logic. The separate views are completely unaware of the controllers handling them. The controllers act as a connection between the model and the view. It handles the inputs from the views by interacting with a facade towards the model and uses the information it receives to update the views. Further details about the connection between the model and the controllers as well as the controllers and the views, can be found in 4.3 and 4.4.

4.1 Subsystem Decomposition

As said before, the application is divided into 4 major packages which can be seen in figure 4. Model, Controllers, Views and Services

Model contains all the data and logic which is used to run the application.

Controller handles inputs and requests from view and communicates with the model.

Views handles the inputs from the user and forwards this to the controller.

Services are used to store data between sessions and to increase the security of the application.

4.2 Model

The model component is responsible for all the data and logic of the program. The top level of the model contains the ChatFacade and four packages that are called server, client, chatcomponents and identifiers.

The model package at the top in figure 4.2 is really just the facade of the model, called ChatFacade. This is responsible for providing a facade for the model package, to minimize the dependencies on model implementations.

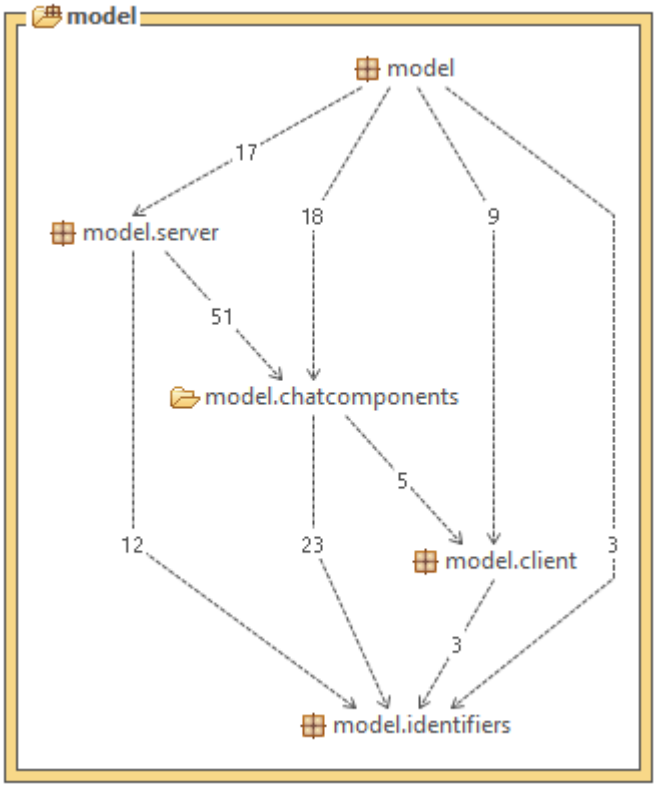


Figure 12: The top level

The client package is responsible for being able to receive updates and provide an interface for anyone that want to add themselves as listener to also receive the updates the client receives. The client will then forward it to all it's listeners.

The "chatcomponents" package is responsible for providing the objects that creates the structure of the chat system. This package is divided into three subpackages, channel, messages and user.

The channel package is responsible of providing a component where several users can send messages to each other.

The message package is responsible for a component that can be sent to others. The message package provides a factory for creating messages so other packages don't need to have dependencies on several different content classes.

The user package contains all data about the user. This package is dependent on the client package because it needs to be able to tell clients when a channel has been updated. Hence, when the user receives an update it will forward these to all it's clients. A user can have multiply clients, to support the ability to log in on the same user on different devices(or in our case, different windows).

The server package is responsible for keeping track of all channels and users. Every time a channel or user is created it is added to the servers lists. The server is also responsible for saving the data when the program closes. To do this, it uses some given data handler that implements IDataHandler. It gives this datahandler certain data objects that stores all the channel or user data as strings. When initializing the server, the given datahandler is used to get these data objects back and then the server turns them into real channel and user objects. Read more about this in section 4.5

By breaking out the Identifiable and Recognizable interfaces into a separate package called identifiers, responsible for providing abstractions of the chat components, the amount of dependencies has been reduced. Furthermore, there were some circular dependencies that have been removed completely due to

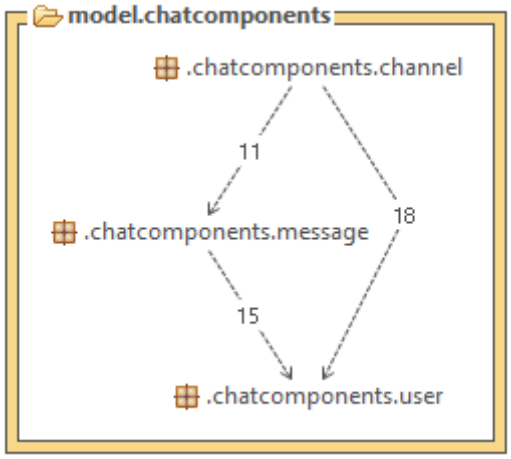
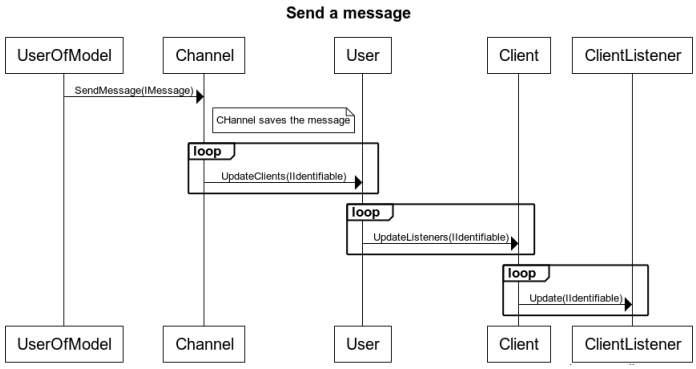


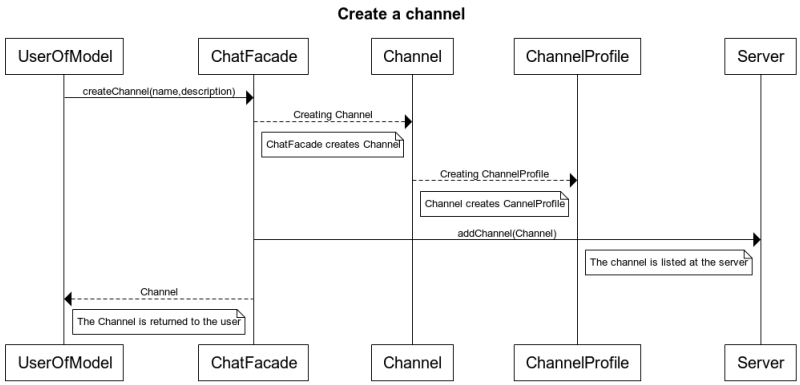
Figure 13: The chat components package

the package decomposition

Below is two sequence diagrams that explains how some processes in the model works.



A channel is used to send a message. The channel saves the message, and notifies any members that the channel has been updated. The members then notifies any clients that a certain channel has been updated. The clients then notifies any client listeners about the update.



As seen in the figure, the ChatFacade will create the channel, add it to the Server, and then return it to be used.

4.3 Controllers

This component takes care of all the communication between the view and the model. It handles all input the view package receives, communicates with the model, and updates the view if needed. This package also listens to the model to be able to update when a channel of the represented client receives a message. To do this the observer pattern is used, and when the model tells this package that a new message has been sent, it will ask the model for the new message, and update the view according to the change.

The controllers are structured in the way that the login view has its own separate controller and the actual chat window has a main controller, that uses several other smaller controllers which each is responsible for a special part of the window.

The controllers are dependent on the model's ChatFacade and the following interfaces:

- IChannel
- IUser
- IClient
- IClientListener
- IIdentifiable
- IRecognizable

Since the model isn't depending on the controllers, it's really easy to change a controller. And since the dependency the controllers has on the model is quite loose (depending only on interfaces and the facade), changes in the implementation of the model can be done without affecting the controllers.

Furthermore, the controllers connection with the view is also quite loosely coupled. The view package provides interfaces for every view component, as well as a corresponding controller interface, that controllers needs to implement to be able to be the controller of a view component.

A controller can then use the view components factory to create such a view and add themselves as the controller, hidden behind an interface. This means that the controller package only is depending on a factory and interfaces, which means that it's easy to change a controller.

If you were to switch one of the controllers, you would only need to make sure it would implement the controller interface of the view component.

4.4 Views

This component is responsible for providing the interface of the program. This view package uses JavaFx and provides several different view components and a corresponding controller interface to each component, and are to be used by any controller that implements the right interface. The views are built in fxml and are dynamically connected with the view classes.

As said in the controller package, the connection between the controllers and views is loose. Hence, if needed, it is quite easy to change the graphic library that are used. If you were to switch library you would only have to create new view components that implemented the view interfaces and used the controller interfaces. In figure 14 you can see the principal of how each controller and view are connected.

4.5 Services

The services package consists of two subpackages, password and datahandler.

The datahandler package is responsible for saving the programs data. To do this, it provides a concrete data handler class which implements an interface in the model. See figure 15. The controller package can then tell the model that this is the data handler to use. The model will then use this data handler, hidden behind the interface, to save data to a file and then later restore the data when its needed. When

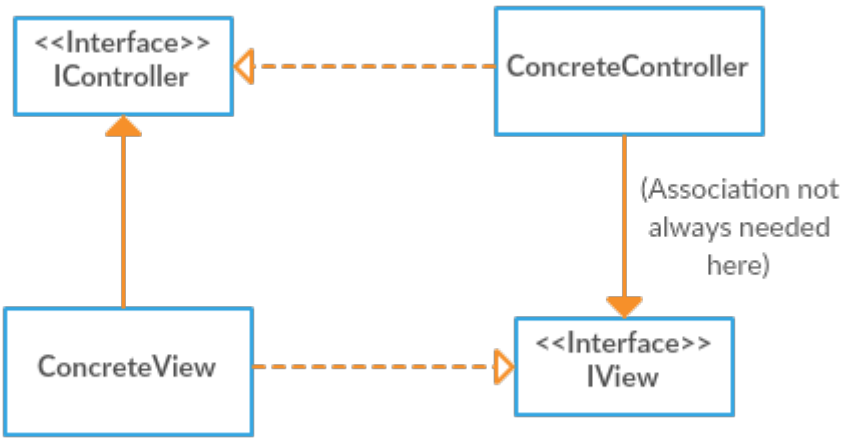


Figure 14: Dependencies between views and controllers

saving, the data handler is given certain data objects, that contains all the data of the components that are to be saved, as strings. When restoring, the model will ask the data handler to represent the data as these data objects and return them so that the model is able to recreate these data objects.

The passwordEncryption package is responsible for making sure to make the passwords encrypted, so when the data handler saves the password, it isn't in plain text. The connection with the model uses the same principal as the datahandler. It provides a concrete password encryption class that implements an interface in the model package. The controller package can then tell the model that this is the password encryptor to use. The model will then use this password encryptor, hidden behind the interface, to hash the users password.

5 Persistent Data Management

The application stores data by using JSON. When closing the application the program creates a JSON object where all the wanted data is stored. The object is then flushed using a FileWriter to a JSON-file. As the data is stored in a JSON file, it means that when we start the application again we can loop through the file and collect the saved data from the previous session.

In order to recreate the program's previous state the data is stored in two different Json files. These are explained below:

Channels are stored in one file, in the form of JSON-objects. These objects contain all the information that a channel controls. Each object contains a key and an associated value in the form of a String, JSONObject, or JSONArray. For example the users that are members of the channel are stored as JSONObjects within the channel JSONObject and the messages written in the channel are stored in a JSONArray.

Users are stored in another file, with each user stored in the form of JSON-objects. These objects contain all the information about the users and can be translated into objects that the model can use.

Images are stored locally in folders and are accessed by using the path to the image.

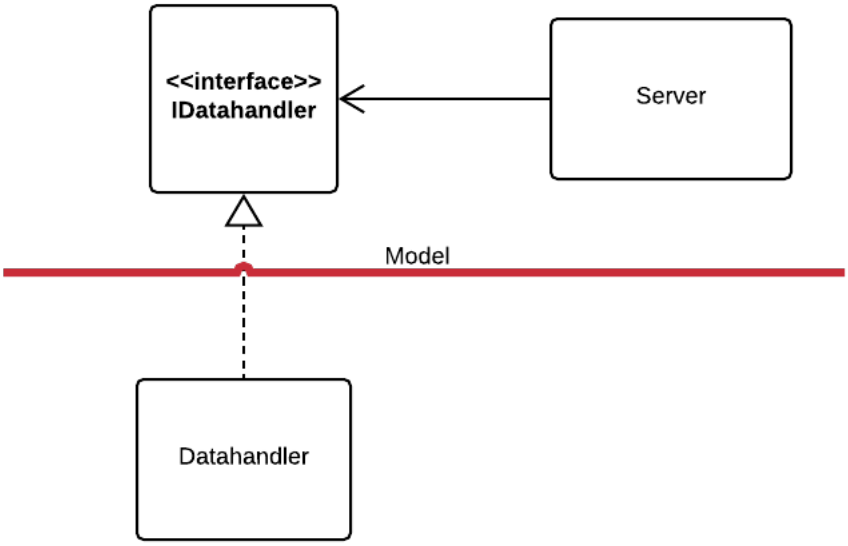


Figure 15: The dependency between the server and the datahandler

6 Access Control and Security

In the application there is different kind of roles which changes some of the things a user can do in the application.

User: The most basic role anyone using the application can have. As soon as you login you become a user. The only thing a user can do is join channels and see other channels.

Channel member: As soon as you join a channel you become a channel member. As a channel member you can write and read messages that are written inside the channel. You also can add other members to the channel.

Channel administrator: If you create a channel you automatically become a channel administrator. A channel administrator can change the channel image, kick members from the channel and also all the functionality a ordinary channel member.

7 Peer review of group19

7.1 Does the project use a consistent coding style?

There is a consistent coding style used throughout the entire model. However, there are certain conventions that aren't used. The member ordering differs between the different classes. For instance, some static/final variables are declared in the middle of the class instead of at the top.

7.2 Is the code reusable?

By looking at STAN we can see that the model is reusable because it doesn't have any dependencies on classes outside it's own package

7.3 Is it easy to maintain?

As the controllers and views are strongly dependent on the model (concrete classes), the code would be quite difficult to maintain as a change in the model probably would require changes to be made in the view and controller as well. A solution would be to have less concrete dependencies so that the views and controllers can depend on interfaces which would allow changes to be made without affecting the views and controllers.

The amount of comments is inconsistent across classes, which sometimes makes it harder for other developers to understand and extend the program. The lifetime would be significantly shorter as new developers find it difficult to maintain.

Some methods in the program are very long which can hurt the maintainability since it will be harder to understand the code and there could be unwanted side effects. An example is the constructor for the UserPageView. It could easily be divided into some smaller methods to make it more maintainable.

7.4 Can we easily add/remove functionality?

We will test this by figuring out how we would extend the application to also support image-messages.

If we for an example are going to implement a way to send image-messages the only way we are going to do this without changing way to much in the existing code is to create a new class that inherit from the Message class. This makes it possible to use polymorphism. However this creates problems later down the road since somewhere in the program we have to use instanceOf to check what kind of message we have sent. It would have been better to create an interface for all the messages which would have made it easier to implement new messages since they would all implement the more abstract interface instead of a concrete class. Using an enum we could also declare the type of the message so we could use that instead of instanceOf.

Overall, we could make it a lot easier to implement new functionality if we depended more on abstract interfaces instead of concrete classes.

7.5 Are design patterns used?

The model uses an implementation of the observer pattern in the model package.

Additionally there is a vague implementation of facade pattern in the mainModel class. However, the facade is a little bit too smart as it handles a lot of data and logic. This leads to the class becoming very large and hard to maintain. Breaking it out into a new class and just creating a ”facade” between mainModel and the rest would be better.

Iterator pattern is also used throughout the mainModel.

7.6 Is the code documented?

Although there are some classes which are fairly well documented (ie mainModel), improvements can be made by explaining what certain methods do rather than just explaining return variables and parameters. There needs to be more documentation in all classes to improve consistency.

7.7 Are proper names used?

Throughout the model the naming conventions are good and easy to understand as methods are very descriptive. However, mainModel is quite a vague name and doesn't really inform new developers of it's purpose.

7.8 Is the design modular? Are there any unnecessary dependencies?

The design is modular as the code can be split up into several different packages. There is an attempt to have high cohesion, low coupling. However, as there are many strong dependencies between packages, the modularity is weakened. This can be improved by splitting up certain classes into smaller classes and trying to reduce the dependencies between the controllers, views, and the model.

7.9 Does the code use proper abstractions?

The controller and view packages does use abstractions well, with several interfaces. However, whenever a class needs to use the model, it has to use a concrete class.

7.10 Is the code well tested?

The code is well tested in certain parts of the program, but there is a lack of code coverage in general. Although the message class doesn't need to have its own test class, it should still be tested through the other classes by using the getters. Additionally, the userTest class is very thin and should have its getters and setters tested in its own test class rather than testing it from a different test class.

7.11 Are there any security problems, are there any performance issues?

The program has several bugs in different areas. For instance, channels sometimes fail to load correctly when a user logs in. Moreover, if a lof of data is saved, the program takes a long time to load a new view when the user clicks on something.

There is also a security problem. If the admin kicks a participant, the participant will still be able to write and see new messages.

7.12 Is the code easy to understand? Does it have an MVC structure, and is the model isolated from the other parts?

Yes, the model is isolated and the application has an MVC structure, although the controller package and the view package have dependencies on each other. This will make it difficult to change just the view package or just the controller package.

The controller does also have a strong coupling with the model, since it's depending on concrete classes. If you were to change some implementations in one of these concrete classes, it would probably affect the controller.